Memory Protection

Peter J. Denning

21 July 1988

RIACS Technical Report 88.17

NASA Cooperative Agreement Number NCC 2-387

RIACS

Research Institute for Advanced Computer Science

Memory Protection

Peter J. Denning

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.17
21 July 1988

Accidental overwriting of files or of memory regions belonging to other programs, browsing of personal files by superusers, Trojan horses, and viruses are examples of breakdowns in workstations and personal computers that would be significantly reduced by memory protection. Memory protection is the capability of an operating system and supporting hardware to delimit segments of memory, to control whether segments can be read from or written into, and to confine accesses of a program to its segments alone. The absence of memory protection in many operating systems today is the result of a bias toward a narrow definition of performance as maximum instruction-execution rate. A broader definition, including the time to get the job done, makes clear that cost of recovery from memory interference errors reduces expected performance. The mechanisms of memory protection are well understood, powerful, efficient, and elegant. They add to performance in the broad sense without reducing instruction execution rate.

This is a preprint of the column *The Science of Computing* for
American Scientist 76, No. 5 (September-October 1988).

Work reported herein was supported in part by Cooperative Agreement NCC 2-387
between the National Aeronautics and Space Administration (NASA)
and the Universities Space Research Association (USRA).

Memory Protection

Peter J. Denning

Research Institute for Advanced Computer Science

21 July 1988

Your program contains an error that overwrites an important input file during a test. Another error makes it calculate an index outside the subscript range of an array and incorrectly read a value from storage allocated to another array. A program running concurrently with yours accidentally writes into your memory region, destroying your program and data files. An entry in a routing table is corrupted by a power failure at a network switching node; copies of the corrupted data propagate throughout the network, producing a crash that leaves you isolated. A system programmer with superuser privileges browses your private, read-only mail files. A program you borrow contains a Trojan horse that steals or erases some of your files. A program you obtained from a network bulletin board contains a virus that attacks your operating system and later erases some of your files.

These nightmarish examples illustrate breakdowns that can occur in the operation of computers when memory protection is inadequate or missing. Memory protection is the capability of an operating system and supporting hardware to delimit segments of memory, to control whether segments can be read from or written into, and to allow access to segments of a program rather than the whole of it. The protection can be extended to files stored on disks.

Memory protection exemplifies the principle that programs should operate with the least privilege required to perform their tasks. Where memory protection prevails, a program cannot write into an input file, refer outside a segment containing a particular array, or refer outside its own set of segments. A power failure cannot write into a read-only table, a superuser cannot automatically override another user's access specifications, a suspected Trojan horse cannot access outside a limited set of segments during a test, and a virus cannot write a copy of itself into any of an operating system's segments.

The need for memory protection was apparent to the early designers of computer systems. In the late 1950s, the Atlas computer at the University of Manchester allowed several independent programs to reside simultaneously in the main memory and included mechanisms that prevented their interfering with each other. Early time-sharing systems experimented with paged virtual memory (IBM 360/67 in 1964), file access protection (MIT's CTSS in 1965), virtual machines (IBM M44/44X in 1966), segmented virtual memory (Honeywell Multics in 1968, Burroughs B6700 in 1970), and capability-based addressing

(Cambridge CAP computer in 1977). The concepts and design principles of these early systems have been incorporated in today's large mainframe operating systems. Excellent accounts of the various developments are provided have been given by Jerry Saltzer and Michael Schroeder (*1*), Maurice Wilkes (*2*), Elliott Organick (*3,4*), and Robert Fabry (*5*).

Few of the penetration and virus problems reported in the news would be possible in machines that use these old memory protection principles (*6*). Yet many of the small operating systems today, such as UNIX™ and the PCs in widest use, contain relatively few or none of the memory protection mechanisms and are easily subverted. Many of the microprocessors in modern workstations and PCs, such as the Intel 80386 and Motorola 68000, contain the requisite hardware, but the operating system simply ignores it. How has this situation arisen?

A major reason for the disappearance of the old concern about memory protection is a bias toward performance, defined as the sustained instruction-execution rate of a machine. Designers of operating systems and hardware simply omit any mechanisms that do not, in their assessment, contribute directly to performance in this narrow sense. Not only do memory protection mechanisms not increase the instruction-execution rate of a machine, but in the poorer implementations they noticeably retard it.

In the past several years, hoowever, a major change in attitude has taken place among computer users. People regularly trust valuable information to

computer files, and years of work can be destroyed in a few milliseconds. It is now becoming fashionable to demand that computer systems be dependable and trustworthy as well as fast. When performance is defined in a broad sense -- minimizing the time to get a job done -- the possible loss of files and subsequent delays in recovering from the loss can be seen as a serious degradation of expected performance.

In what follows I will describe the main parts of a memory protection mechanism in an attempt to convey the elegance and power of the design. As we will see, there need be no significant loss of speed, because most of the access checking can be done by the hardware in parallel with the main computation.

Most computers consist of processors, fast main memory, and disk memory. The addressing interface between processor and main memory allows for two types of commands, "read A " and "write A ." To read, the processor places address A into the address register and signals the memory; the memory copies the value from address A into the data register and returns an acknowledge signal to the processor. To write, the processor places a value into the data register and address A into the address register, then signals the memory; the memory replaces the contents of address A with the value and signals an acknowledgment to the processor. If the memory detects an error during either of these operations, it returns a fault signal rather than an acknowledgment to the processor.

Most computations are composed of parts -- the various program modules and data files -- which are stored in the disk memory as files. When one of these parts is loaded into main memory, it is called a segment. A segment can be described with two numbers, B and L ; B is the base or starting address of the segment, and L is the length; the segment occupies addresses B , $B+1$, $\cdots$, $B+L-1$.

When a segment is present in main memory, its descriptor (B ,L) can be stored in a register within the addressing mechanism and used in the following way. The read and write commands are automatically interpreted relative to the base address in the descriptor register; thus "read A " is actually applied to address $B+A$. While forming address $B+A$, the memory interface also checks that A is within the span of the segment -- that is, that $0 \leqslant A < L$; if not, a range error is signaled, and the processor is not allowed to complete the access to address $B+A$.

The descriptor can be extended to include a presence bit P , set to 1 when a copy of the segment is loaded in main memory and otherwise to 0. If the segment is marked as not present the hardware signals a missing segment fault that interrupts the addressing process and brings into execution a program to locate the file in the disk system, load it, and set the presence bit of the descriptor to 1.

This basic mechanism provides a processor with read and write access to exactly one segment, a mode of operation useful when all parts of a computation are stored in a single segment. It prevents a processor from reading or writing

regions of memory assigned to any other computation. In practice, however, there is a need to protect the separate parts of a computation – for example, when one of the parts is a borrowed program that might contain a Trojan horse or is a new program module that has not been fully tested. The mechanism must be refined to allow a set of several descriptors to be associated with a computation.

To do this, each descriptor is marked with a unique key, and all the keyed descriptors are stored in a table. When given a key, the addressing hardware searches the descriptor table for an entry with that key. If the search succeeds, the particular P , B , and L values are indicated as above. High speed associative memories can be used to hold the descriptor tables so, that the search time is small compared to the time of a read or write command. Note that the keys do not change as the segment is moved between the main and disk memories (differing presence bit values), relocated within the main memory (differing base values), or altered in size (differing length values). Thus a program's design is independent of the details of physical storage, and a major source of programming errors is avoided (*7*).

How does a computation generate the keys corresponding to the various segments? Associated with the computation is a segment table that contains a list of segment accessors, each of which consists of an access code (whose bits enable reading and writing) and a key that matches one of the descriptors. To use this table, the processor must specify the segment within which a given

address falls; thus "read N , A " means read the value in address A of segment N . Using the accessor at position N of the segment table, the addressing hardware checks that reading (or writing) is permitted by the access code and if so initiates the search of the descriptor table; the rest is the same as the process of using a descriptor outlined above. As before, high speed associative memory can be used to keep these searches to a negligible fraction of the time for reading and writing. The full mechanism is summarized in the accompanying box.

This design handles shared segments elegantly. There is no requirement that the computations sharing a given segment assign the same segment number to it. Program code modules, which are read-only, are the most common candidates for sharing. All the user sessions employing the same text editor, for example, can have accessors containing copies of the key for the descriptor of the editor's instruction code segment. The editor obtains access to text files by referring to separate segments, and the numbers of those segments can be passed to the editor as parameters. Because it is not necessary to have more than one copy of the editor's code segment loaded in main memory, substantial savings of memory are possible in multi-user systems.

A simple extension of this mechanism increases its power substantially. A computation involving a set of segments can be regarded as a unit. For example, the set of programs for opening, closing, reading, and writing files can be encapsulated in a package along with segments containing private information about the status and location of files in the disk storage system. A call on any

one of these four programs automatically switches the computation so that they operate with their own segment table, which is distinct from the segment table of the caller. The ability to encapsulate a package substantially increases its reliability, because none of the private data can be consulted or modified by any programs other than those authorized to do so. Such a mechanism should prevent viral infection of package components.

To implement this extension, a computation number C stored in the processor is used to inform the addressing mechanism which segment table should be used. An accessor for C consists of a code enabling "call" and the key "C." A segment number, say N, corresponding to an accessor for C is allocated in the segment table of a computation authorized to call C; then the command "call N" will automatically invoke the entry procedure of C, and the processor will begin using the segment table to C. Fabry gives a full account of this mechanism (5).

Where do the entries in the segment tables come from in the first place? They come from the file system. Each user owns a tree of directories. Each directory, stored as a file on a disk, contains entries that point to other files by giving their unique keys. Each entry also contains a field that points to an access control list specifying names and access codes for that file. Thus, when one of my computations attempts to open a file and load it into a memory segment, the operating system checks whether my name is on the access list of that file; if so, it creates an accessor with that code whose key is the same as the file's

unique key.

I have only scratched the surface of what can be done with the design outlined here. My intention has been simply to exhibit the design principles of memory protection, and to suggest that they can be implemented with negligible impact on instruction execution speed. The added benefits -- significantly increased protection against faults and errors, reduction in exposure to Trojan horses and viruses, and resistance against unauthorized access to one's files -- are well worth the additional design expense.

Scientists and engineers should be much more forceful in expressing their concerns about memory protection, so that future generations of workstations and personal computers willbe dependable and secure as well as fast.

References

1. J. Saltzer and M. Schroeder. 1975. "The protection of information in computer systems." *Proc. IEEE 63*, 9. September. 1278-1308.

2. M. V. Wilkes. 1975. *Time Sharing Computer Systems.* Elsevier North Holland. 3rd Edition.

3. E. I. Organick. 1972. *The MULTICS Systems: An Examination of Its Structure.* MIT Press.

4. E. I. Organick. 1973. *Computer System Organization: The B5700/6700 Series.* Academic Press.

5. R. Fabry. 1974. "Capability-based addressing." *Communications of ACM 17*, 7. July. 403-412.

6. P. J. Denning. 1988. "Computer viruses." *American Scientist 76*, 3. May-June. 236-238.

7. P. J. Denning. 1986. "Virtual memory." *American Scientist 74*, 3. May-June. 227-229.

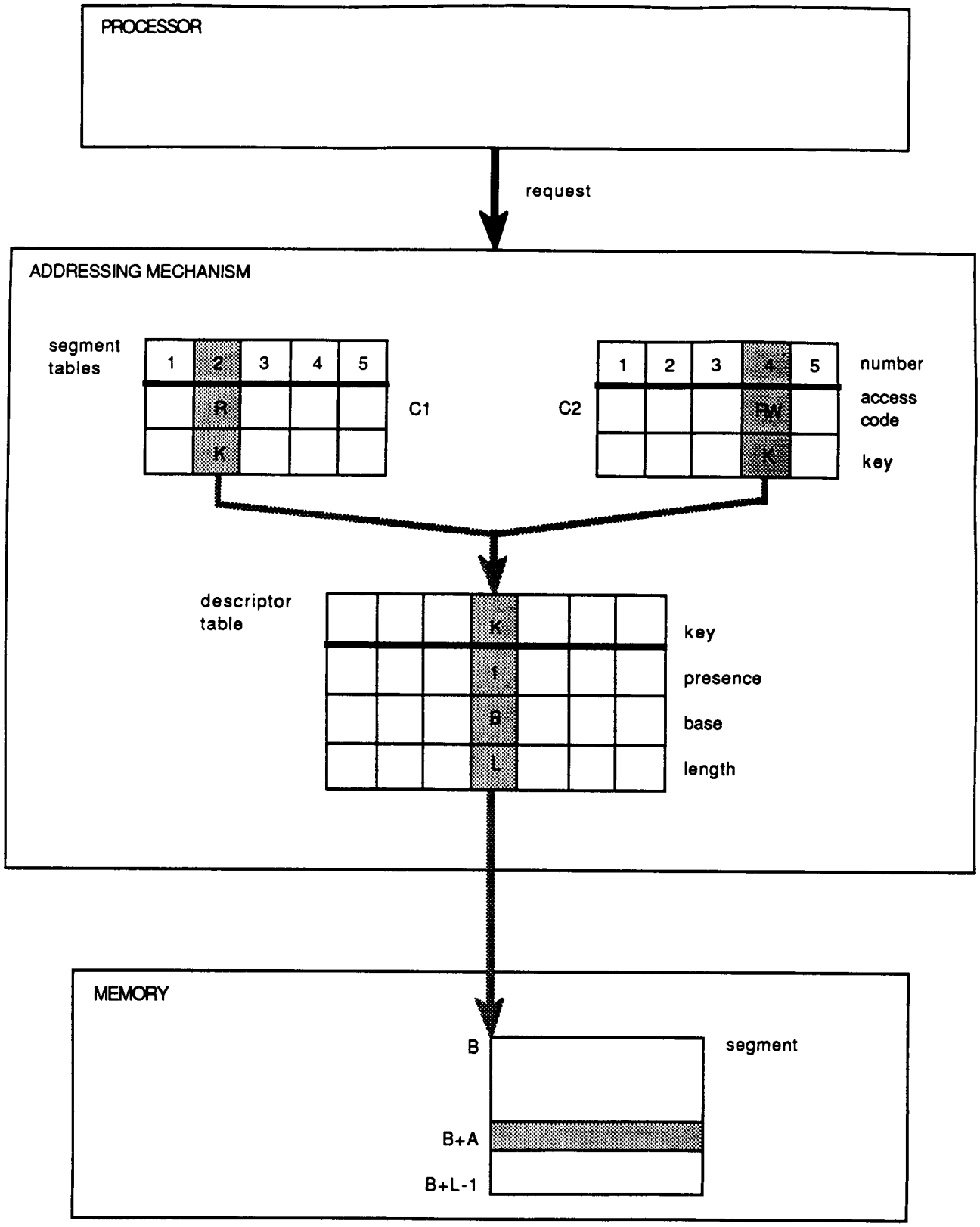
Addressing for Memory Protection

The addressing mechanism for protected segments contains two levels of mappings that transform a request like "read address A of segment N of computation C" into a reference to the proper memory location $B+A$.

The upper level consists of segment tables, such as $C1$ and $C2$, attached to computations themselves; it associates a segment number N (1, 2, 3, 4, ...) with an access code and unique key for each segment. Access code R in the figure enables reading, and access code RW enables both reading and writing. A shared segment with key K can have different numbers and different access codes in different computations.

The lower level consists of a descriptor table attached to the memory; it records presence, base, and length descriptors for each segment. The presence bit P is 1 if a copy of the segment is in main memory; if $P=0$, the segment is in disk memory and must be located and loaded in main memory before it can be used. The base B gives the starting address of a loaded segment. The length L gives the number of consecutive addresses occupied by the segment. The last address in the segment is thus $B+L-1$. Segments can be moved between levels of memory, relocated within main memory, or changed in length without changing segment numbers, access codes, or keys or otherwise disturbing any component of a computation.

To read from a segment, a processor operating on behalf of computation C generates a request "read N,A." The addressing mechanism looks at entry N in the segment table for C; if read access is permitted, it searches the descriptor table for K; it then requests memory to read from $B+A$, provided that $0 \leqslant A < L$. These steps can be completed in a negligible fraction of the time the memory takes to read. If any of the checks embedded in them fails, the addressing system stops the processor with an error signal.


PROCESSOR
request
ADDRESSING MECHANISM
segment tables
1
2
3
4
5
R
K
C1
C2
number
access code
key
descriptor table
K
f
B
L
presence
base
length
MEMORY
B
B+A
B+L-1
segment